CMPS 312
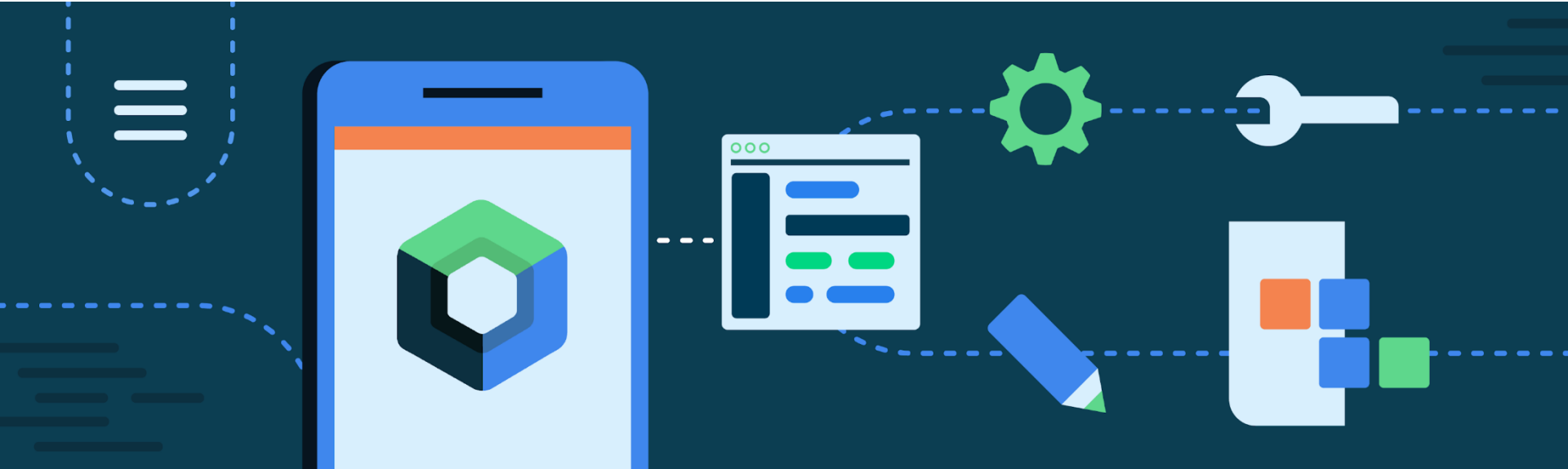

Declarative UI using Jetpack Compose

Dr. Abdelkarim Erradi

CSE@QU

Outline

1. [Jetpack Compose Key Concepts](#)

2. [UI Components](#)

3. [Modifiers](#)

4. [Layouts](#)

5. [State](#)

Jetpack Compose Key Concepts
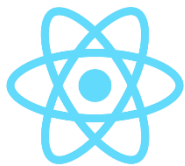
Declarative UI is a major trend 📈

- Describe WHAT to see NOT HOW

[Flutter](): Google's UI toolkit for building natively compiled applications for mobile, web and desktop from a single codebase

[SwiftUI](): Apple's new declarative framework for creating apps that run on iOS

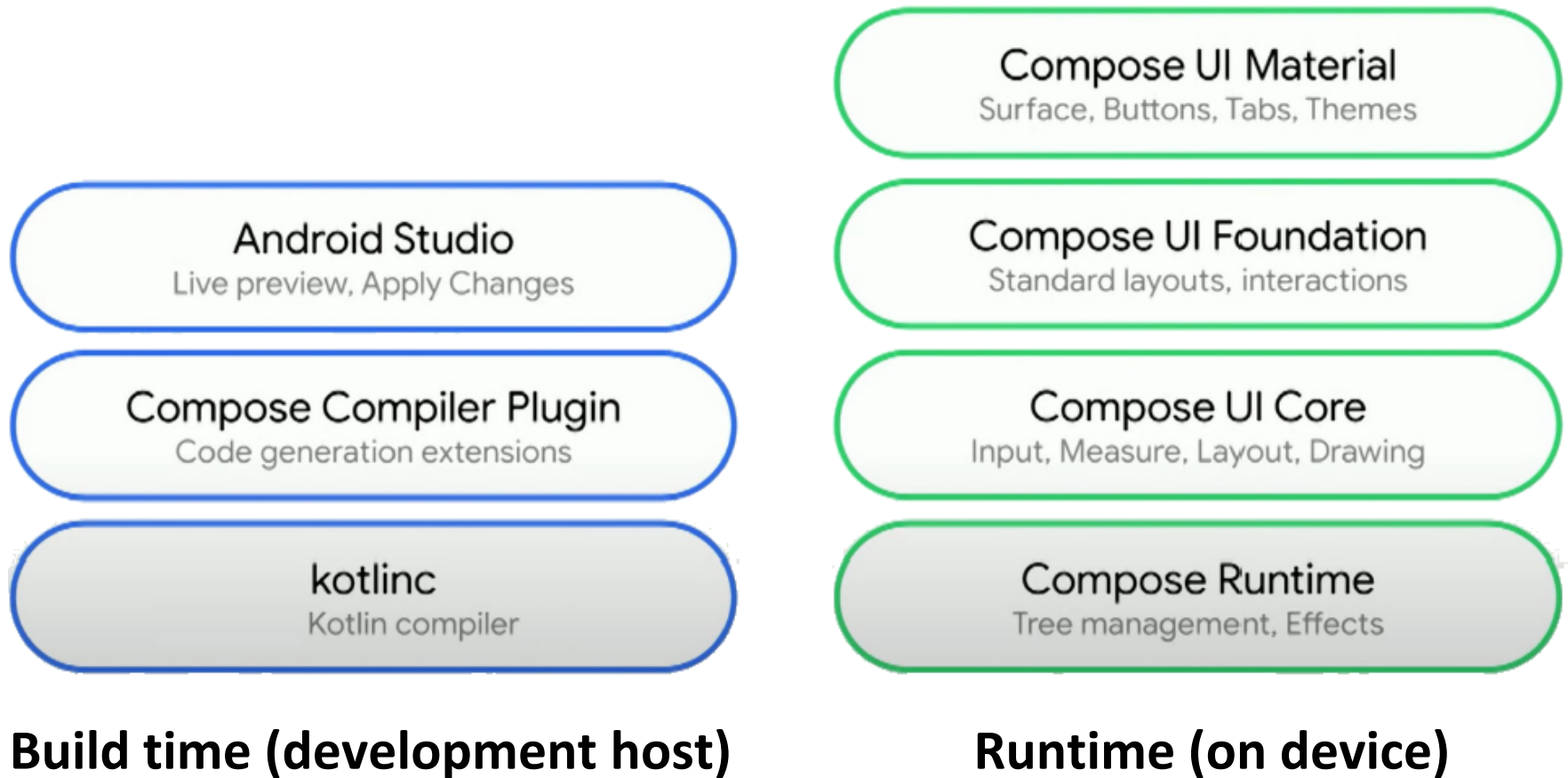[React](): A JavaScript library for building user interfaces

[Jetpack Compose](): a **modern toolkit** for building native Android UI ([released July 2021]())

Jetpack Compose

- Jetpack Compose is a **modern toolkit** for building native Android UI

  - It simplifies UI development with less code and intuitive Kotlin APIs that follow best practices

- A **declarative component-based programming model**

  - UI is built using composable functions

    - Each function define a piece the app's UI programmatically by **describing WHAT to see** (layout/ look and feel) **NOT HOW**

  - As state changes the UI automatically updates (Reactive UI)

  - Inspired by/similar to other declarative UI frameworks such as React and Flutter

Inside Jetpack Compose

Compose UI Material
Surface, Buttons, Tabs, Themes

Android Studio
Live preview, Apply Changes

Compose UI Foundation
Standard layouts, interactions

Compose Compiler Plugin
Code generation extensions

Compose UI Core
Input, Measure, Layout, Drawing

kotlinc
Kotlin compiler

Compose Runtime
Tree management, Effects

Build time (development host)   **Runtime (on device)**

What's new in Jetpack Compose (Android Dev Summit '19)

🧑‍🎨 How to define a piece of UI?

- UI is **composed** of small <u>reusable</u> **components**

- UI Component = Composable **function**:

  - Just a function annotated with **@Composable**

  - Take some <u>inputs</u> and return a piece of <u>UI</u>

  - Describe the UI based on the provided parameters

    - Describes **WHAT to see** NOT HOW (@Composable does the magic for HOW)
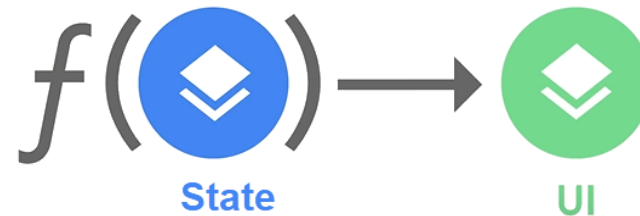
$f(\text{State}) \rightarrow \text{UI}$

  - **UI = f(state) : UI is a visual representation of state**

- 👍 **State changes trigger automatic update of the UI**

UI as a function

Composition

- **UI = Composition of UI functions**
- **UI Function = Building blocks**

```
@Composable
fun WelcomeScreen() {
    var userName by remember { mutableStateOf( value: "Android") }
    Column {  this: ColumnScope
        NameEditor(name = userName, nameChange = { newName -> userName = newName })
        Welcome(userName)
    }
}

@Composable
fun NameEditor(name: String, nameChange: (String) -> Unit) {...}

@Composable
fun Welcome(name: String) {...}
```

Your name
Android

Welcome Android!

Change Color (clicked 9 times)

9

Entry point to Compose world

- When the app launches it creates and starts the Main Activity

- **Activity** acts as a container to load the **main UI screen**

  - o Using **setContent** in the **onCreate** method

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyAppTheme {
                Surface(color = MaterialTheme.colors.background) {
                    Greeting("Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(stringResource(R.string.hello, name))
}
```
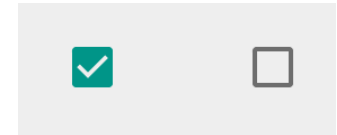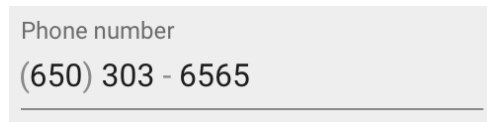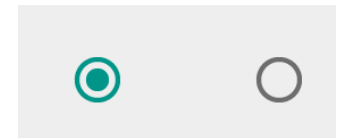
```
23:38   A  Q  O                           LTE
Hello Android!
```

UI Components

Button

CheckBox

EditText

RadioButton

SeekBar

Switch

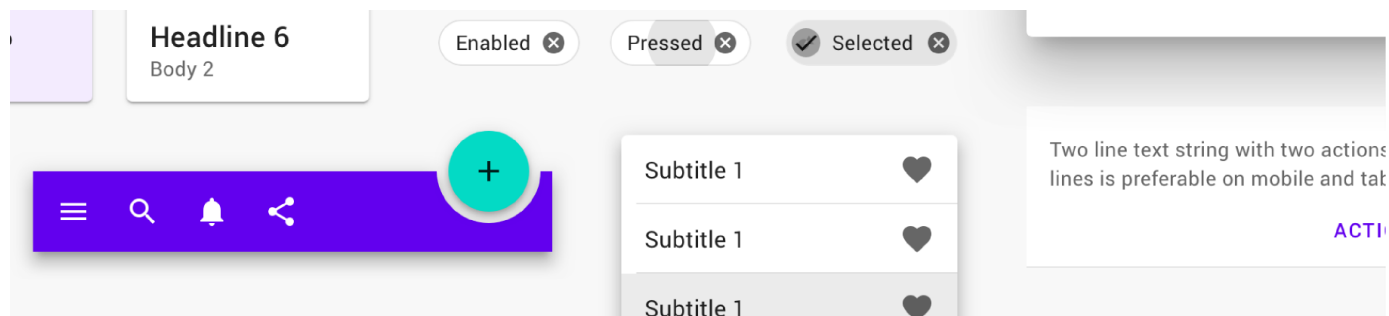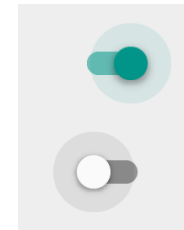**Full list available at this link**

Text box

```kotlin
@Composable
fun Number(value: Int) {
    Text(
        text = value.toString(),
        fontSize = 20.sp,
        modifier = Modifier
            .size(40.dp)
            .background(Color.Black)
    )
}
```

Button

```
Button(
    text = "Button",
    icon: Icon? = myIcon,
    textStyle = TextStyle(...),
    spacingBetweenIconAndText = 4.dp,

    ...

)
```



```
Button(onClick = {}) {
    Text("Button")
}

OutlinedButton(onClick = {}) {
    Text("OutlinedButton")
}

TextButton(onClick = {}) {
    Text("TextButton")
}
```
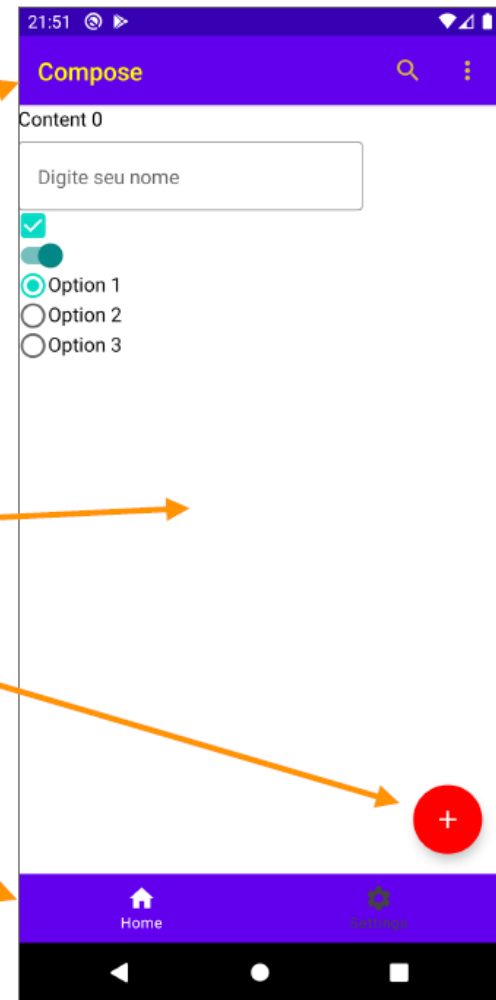
Image

```
Image(painter =
    painterResource(R.drawable.img_compose_logo),
    contentDescription = "Jetpack compose logo",
    modifier = Modifier.sizeIn(maxHeight = 300.dp))
```
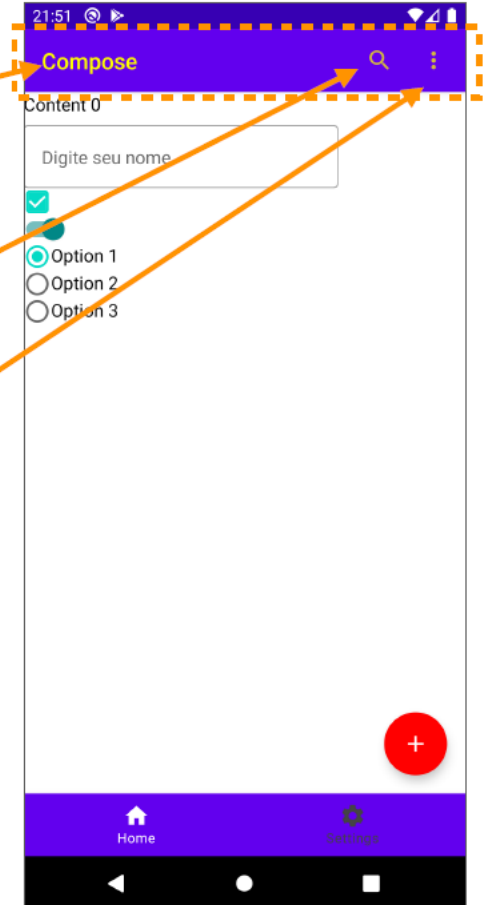
Scaffold


```
Scaffold(
    topBar = {...},
    floatingActionButton = {...},
    bottomBar = {...}
) {...}
```

TopAppBar

```
TopAppBar(
    title = { Text(text = "Compose") },
    backgroundColor = MaterialTheme.colors.primary,
    contentColor = Color.Yellow,
    actions = {
        IconButton(onClick = {}) {
            Icon(Icons.Default.Search, "Search")
        }
        IconButton(
            onClick = { … }
        ) {
            Icon(Icons.Filled.MoreVert, "More")
            DropdownMenu(…)
        }
    }
)
```
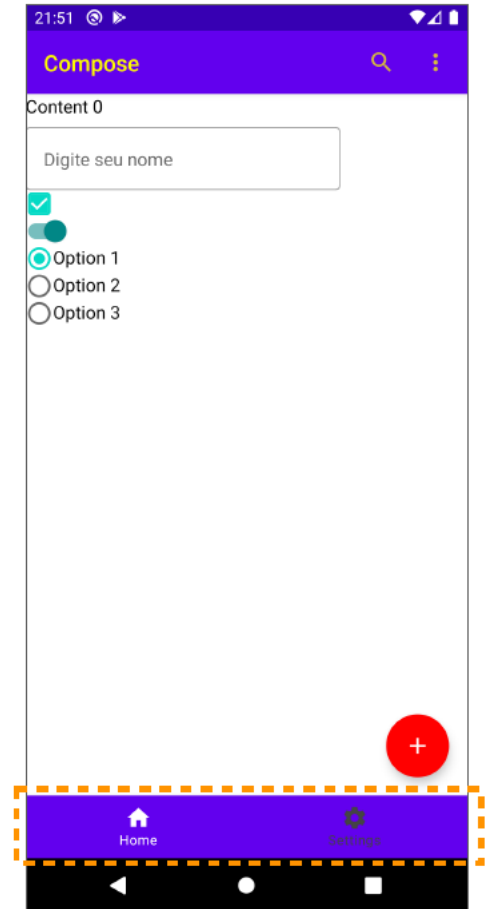
FloatingActionButton

```
FloatingActionButton(
    onClick = { … },
    backgroundColor = Color.Red,
    contentColor = Color.White
) {
    Icon(Icons.Filled.Add, "Add")
}
```
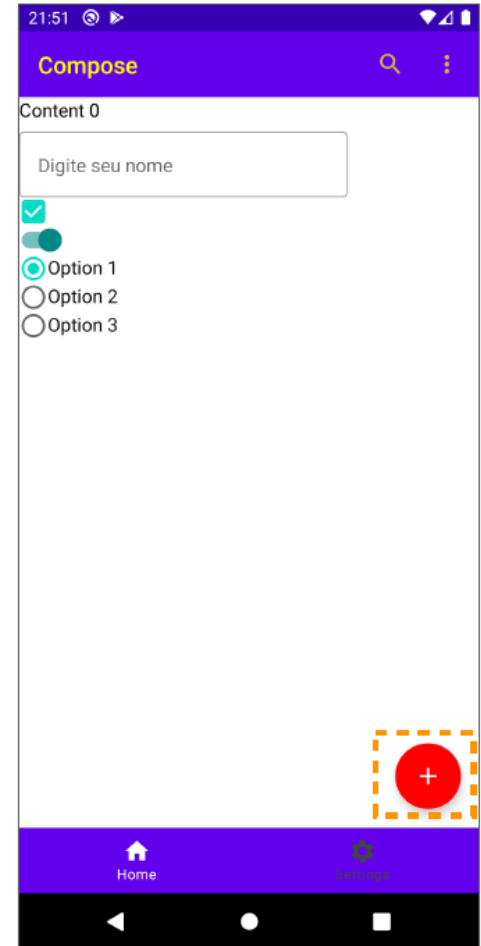
BottomAppBar

```
BottomAppBar(
    backgroundColor = MaterialTheme.colors.primary,
    content = {
        BottomNavigationItem(
            icon = { Icon(Icons.Filled.Home) },
            selected = selectedTab == 0,
            onClick = { selectedTab = 0 },
            selectedContentColor = Color.White,
            unselectedContentColor = Color.DarkGray,
            label = { Text(text = "Home") }
        )
        BottomNavigationItem(…)
    }
)
```

Material Design Components

- Using MDC to make your app look great easily

https://material.io/components

- o Float labels – TextInputLayout
- o FloatingActionButton
- o NavigationDrawer
- o Toolbar
- o CardView
- o TabLayout
- o BottomNavigation
- o BottomSheet
- o Snackbar

Will be used Later

AlertDialog
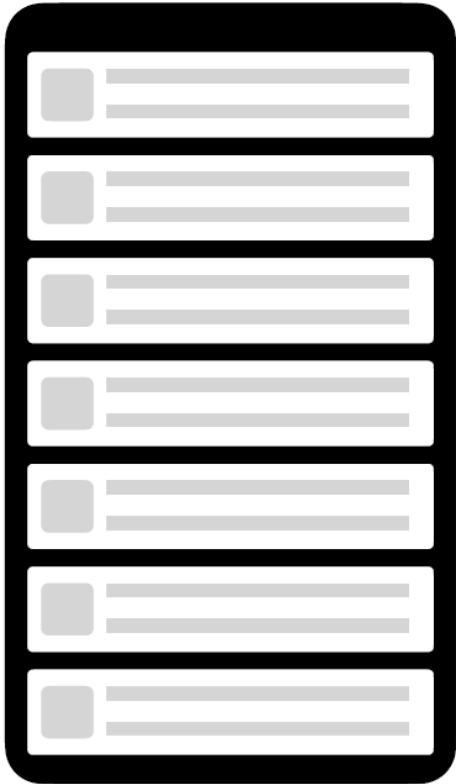
- TBD

Will be
used
Later

List

```kotlin
@Composable
fun SurahsList(surahs: List<Surah>) {
    Column(modifier =
     Modifier.verticalScroll(rememberScrollState())
    ) {
        if (surahs.isEmpty()) {
            Text("Loading surahs failed.")
        } else {
            surahs.forEach {
                SurahCard(surah = it)
            }
        }
    }
}
```

List in Compose (with index)

```kotlin
@Composable
fun UserListScreen(users: List<User>) {
    LazyColumn(
        modifier = Modifier.fillMaxSize()) {
        item {
            Text("Header",
                Modifier.fillMaxWidth().padding(8.dp)
            )
        }
        itemsIndexed(users) { index, user ->
            Text("${user.name} - ${user.age}",
                Modifier.fillMaxWidth().padding(8.dp)
            )
        }
    }
}
```
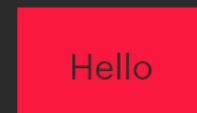
Modifiers

Modifiers

- Modifiers amend the look or behavior of UI components

e.g., provide **spacing**, **layout parameters** and **assign behavior** such as clickable

- They're chained and the order matters!

  o Applied in a sequential way and the order impacts the behavior

```
Text(
    text = "Hello",
    modifier = Modifier.padding(16.dp)
        .background(color = Color.Red)
)
```
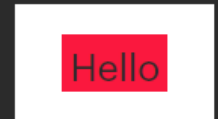
Hello

```
Text(
    text = "Hello",
    modifier = Modifier.background(color = Color.Red)
        .padding(16.dp)
)
```

Hello

Photographer Card

```kotlin
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
    }

        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```
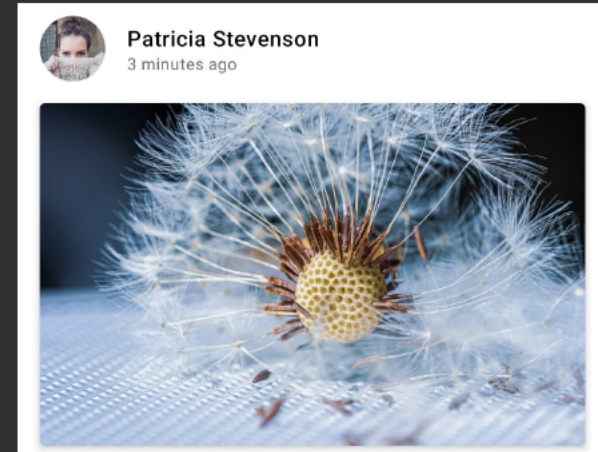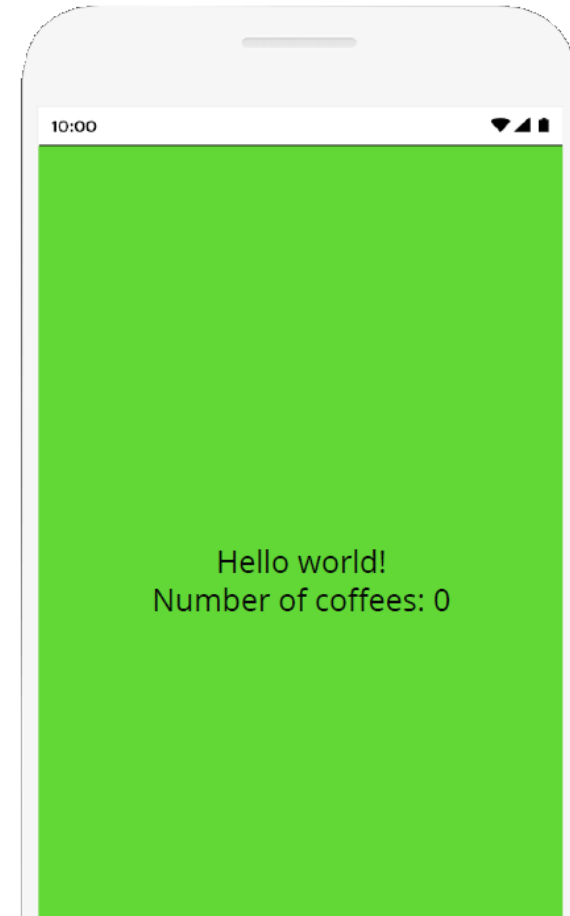
Another Modifier Example

```kotlin
@Composable                                                    Kt
fun Greeting(name: String) {
  Column(
    modifier = Modifier.fillMaxSize()
                       .background(Color.Green)
                       .padding(16.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
  ){
    Text(text = "Hello $name!")
    Text(text = "Number of coffees: 0")
  }
}
```
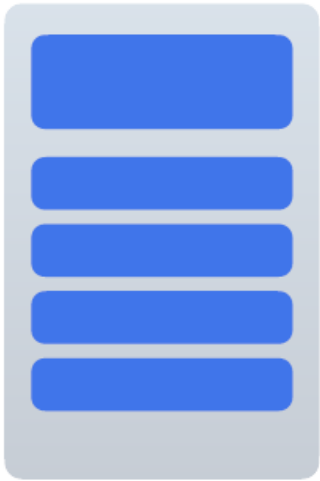
Hello world!
Number of coffees: 0

Layouts
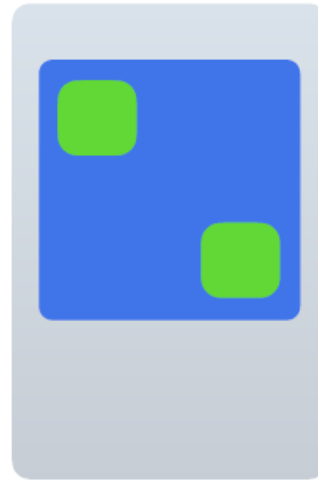
Compose Layout

- Column = vertical orientation
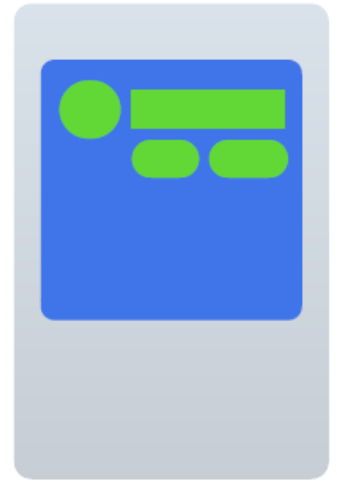- Row = horizontal orientation

Column   Row   Box   Constraint Layout

Row & Column Example

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(/*...*/)
        Column {
            Text(artist.name)
            Text(artist.lastSeenOnline)
        }
    }
}
```
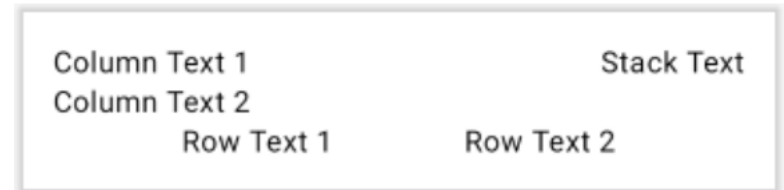
Alfred Sisley
3 minutes ago

Box Example

```kotlin
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(/*...*/)
        Icon(/*...*/)
    }
}
```
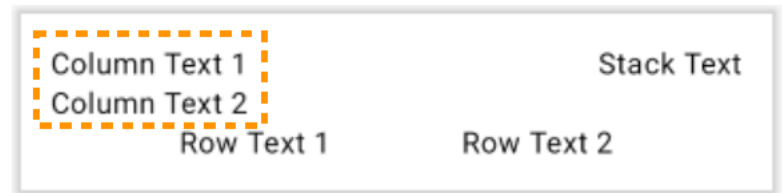
Box Example (1 of 4)

```
Box(modifier = Modifier.fillMaxWidth()) {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Text("Column Text 1")
        Text("Column Text 2")

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly
        ) {
            Text(text = "Row Text 1")
            Text(text = "Row Text 2")
        }
    }
    Text(
        "Stack Text",
        modifier = Modifier
            .align(Alignment.TopEnd)
            .padding(end = 16.dp, top = 16.dp)
    )
}
```

Column Text 1		Stack Text
Column Text 2		
	Row Text 1	Row Text 2

```
Box(modifier = Modifier.fillMaxWidth()) {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Text("Column Text 1")
        Text("Column Text 2")

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly
        ) {
            Text(text = "Row Text 1")
            Text(text = "Row Text 2")

        }
    }
    Text(
        "Stack Text",
        modifier = Modifier
            .align(Alignment.TopEnd)
            .padding(end = 16.dp, top = 16.dp)

    )
}
```

```kotlin
Box(modifier = Modifier.fillMaxWidth()) {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Text("Column Text 1")
        Text("Column Text 2")

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly
        ) {
            Text(text = "Row Text 1")
            Text(text = "Row Text 2")
        }
    }
    Text(
        "Stack Text",
        modifier = Modifier
            .align(Alignment.TopEnd)
            .padding(end = 16.dp, top = 16.dp)
    )
}
```
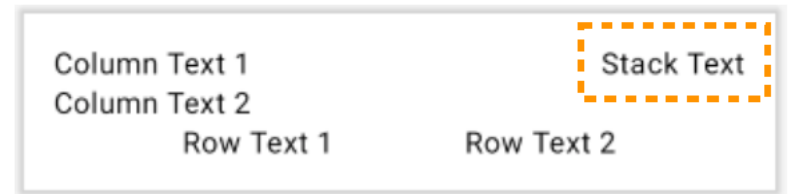
Column Text 1		Stack Text
Column Text 2		
	Row Text 1	Row Text 2

Box Example (4 of 4)

```kotlin
Box(modifier = Modifier.fillMaxWidth()) {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Text("Column Text 1")
        Text("Column Text 2")

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly
        ) {
            Text(text = "Row Text 1")
            Text(text = "Row Text 2")
        }
    }
    Text(
        "Stack Text",
        modifier = Modifier
            .align(Alignment.TopEnd)
            .padding(end = 16.dp, top = 16.dp)
    )
}
```

Column Text 1	Stack Text
Column Text 2	
Row Text 1	Row Text 2

State

State

- State = value that can change overtime
- State variable must be declared as

remember { *mutableStateOf*(...) }

- o **Remember** in the composable memory to hold the state
- o State variable **are observed** by the Jetpack compose runtime
  Any value changed in the state will trigger UI recomposition
  - UI update is handled by the Jetpack compose runtime not by the developer
  - Every place a state variable is displayed is guaranteed to be auto-updated

```
var nameState by remember { mutableStateOf("") }
TextField(
    value = nameState,
    label = { Text("Name") },
    onValueChange = { s: String ->
        nameState = s
    }
)
```

Imperative UI vs. Declarative UI

- ## Imperative UI

```
TextView greetings = (TextView) findViewById(R.id.tv_greeting)
greetings.text = "Hello world."
```

10:00

Hello world.

ANDROID:ID = "@+ID/TV_GREETING"
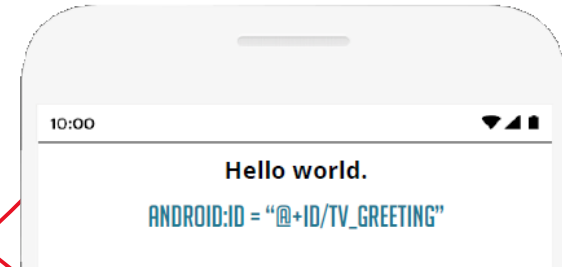
- ## UI in Compose is immutable

  - In compose you cannot access/update UI elements directly (as done in the imperative approach)

  - The only way to update the UI is by updating the state variable(s) used by the UI elements

    - E.g., displayed *greeting text* can only be changed by updating the *name* state variable
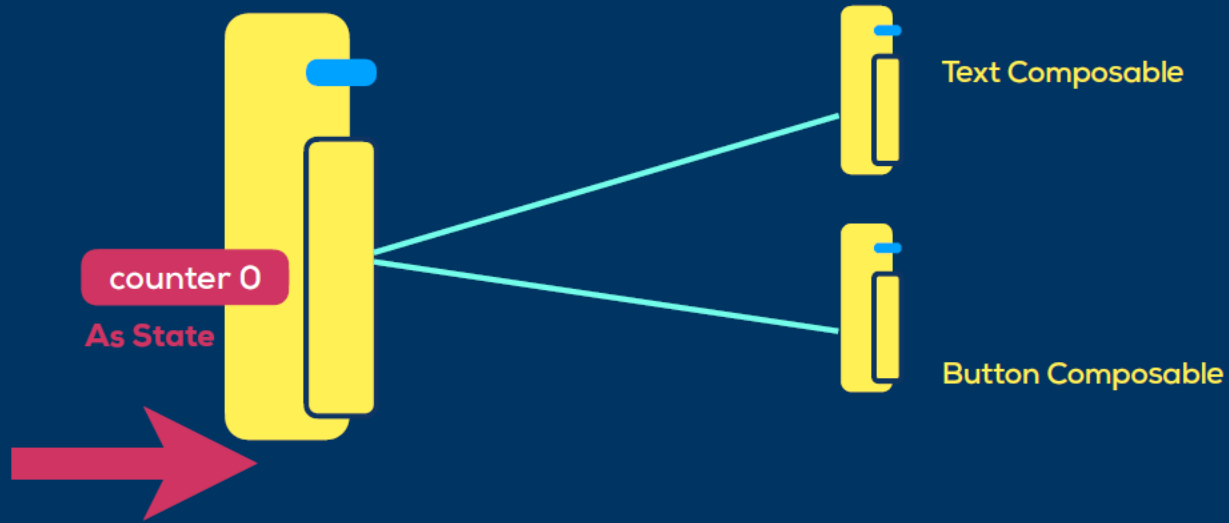
```
@Composable
fun WelcomeScreen() {
    var name by remember { mutableStateOf("Android") }
    Greeting(name)
}
```

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```
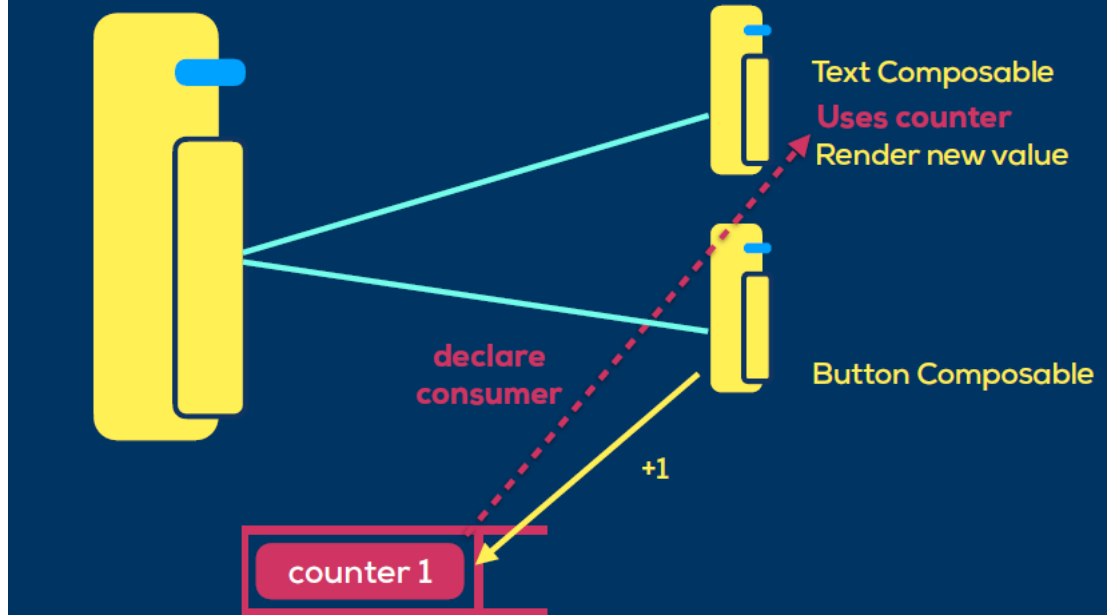
State Hoisting

- Extract the state from the composable, move it to the caller of the composable

- & pass it to the composable as an immutable parameter, along with functions as parameters to represent events
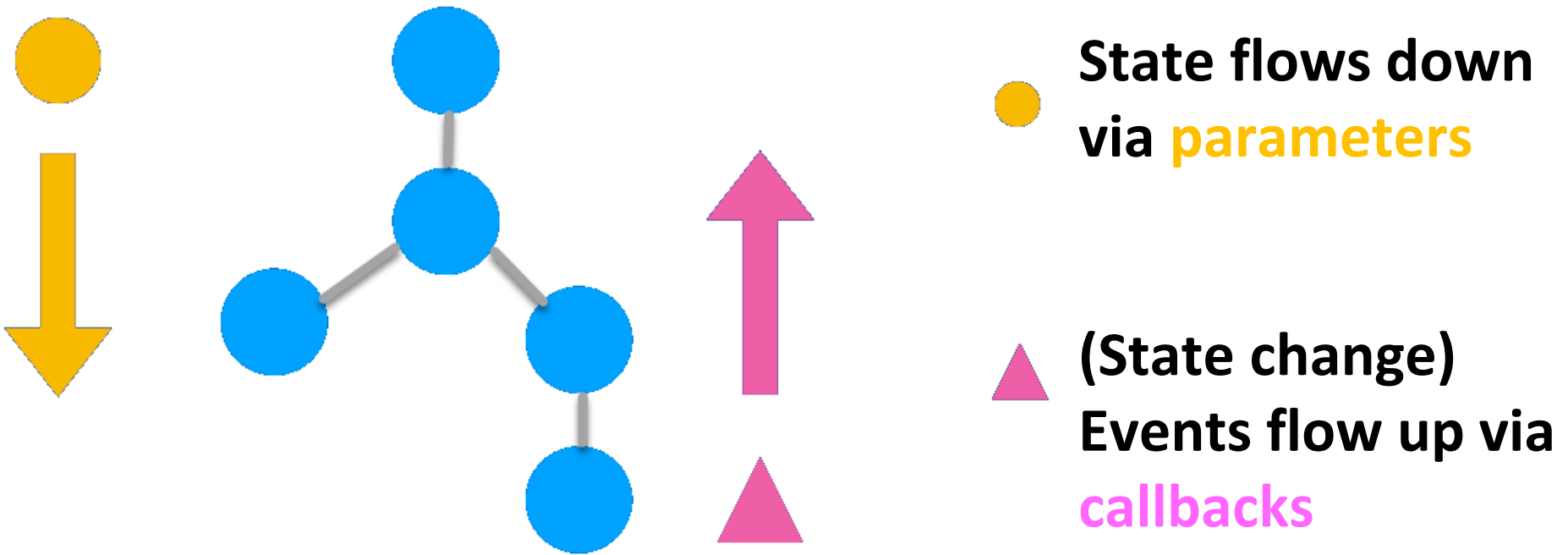
Column Composable

counter 0
As State

Text Composable

Button Composable

Recompose

Column Composable

Text Composable
Uses counter
Render new value

declare
consumer

Button Composable

+1

counter 1

Unidirectional Data Flow

State flows down via **parameters**

(State change) Events flow up via **callbacks**
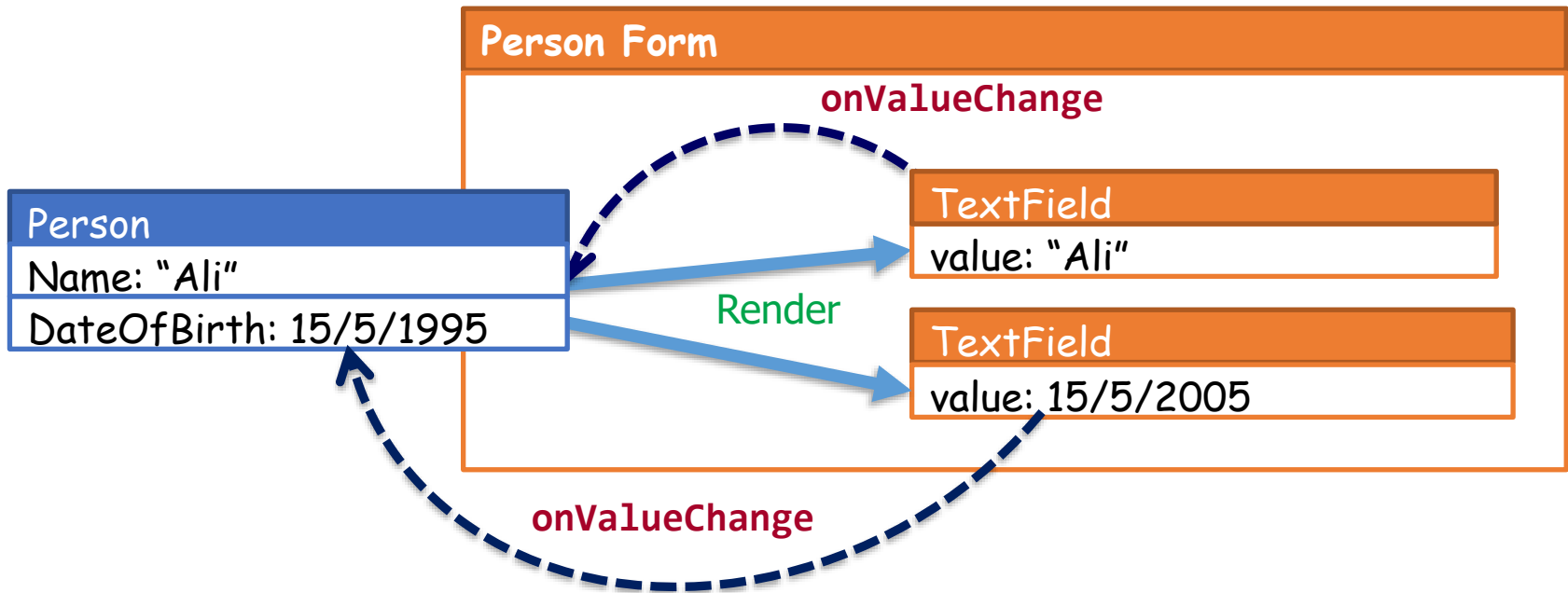
Unidirectional Data Flow

Recomposition

- In an imperative UI model, to change a view, you call a setter on the view to change its internal state.

- In Compose, you call the composable function again with new data. Doing so causes the function to be recomposed--the view emitted by the function are redrawn, if necessary, with new data.

  - The Compose framework can intelligently recompose only the components that changed

How recomposition works

1. Creates an abstract representation of the UI and renders it

2. When a change occurs, it creates a new representation

3. Computes the differences between the two representations

4. Renders the differences [if any]
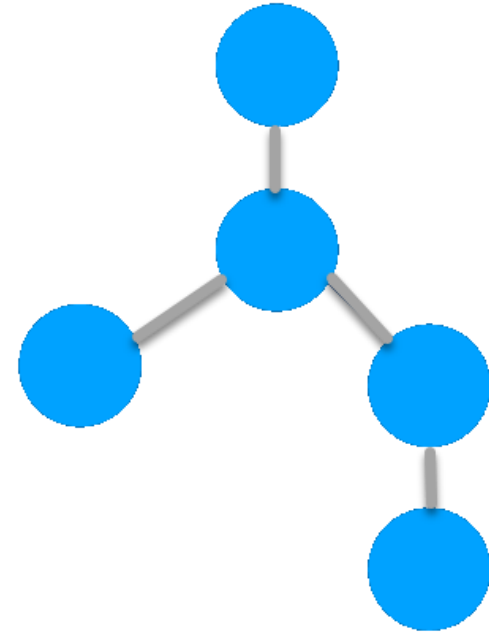
For more details about Jetpack Compose Runtime, watch this video

Summary

- Declarative UI is the trend for UI development
- UI is composed of small <u>reusable</u> components
- UI Component = Composable function
- UI in Compose is immutable
- It only accepts state & expose events
- Unidirectional data flow pattern:
  - State flows down via parameters
  - Events flow up via callbacks

Resources

- Jetpack compose tutorial

https://developer.android.com/jetpack/compose/tutorial

- Jetpack compose Code Labs

https://developer.android.com/courses/pathways/compose

- Compose Samples

https://github.com/android/compose-samples